

Modified Counting Sort



Ravin Kumar

Abstract There are various sorting methods in the literature, which are sequential in nature and have linear time complexity. But these methods are not preferred to use due to large memory requirements in specific cases. Counting sort is one, which lies in this domain. In this chapter, we have suggested an improvement on the counting sort. Due to this improvement, the memory requirement for counting sort is reduced up to a significant level. We have tested this modified counting sort on numerous data sets and the results obtained by these experiments are very much satisfactory. Results shows that this memory requirement is reduced at least 50% than traditional counting sort. So it opens up the opportunity of using this modified version in many sorting applications.

Keywords Sorting • Linear sorting • Counting sort • Modified counting sort
Non-comparison sorting

1 Introduction

Sorting is the process of arranging numbers, alphabets or characters in a statistical order (in increasing or decreasing order) or in lexicographical order (alphabetical value like addressee key) [1–3]. These techniques are divided into categories based on their features. Research efforts are made to improve their performance by reducing their complexity. These algorithms are mainly divided into two categories, comparison-based sorting and non-comparison-based sorting. Comparison-based sorting algorithms involve comparing the values in the array to obtain the sorted sequence. Depending on the type of approach used to perform comparisons, few mainly used algorithms with their complexities are as stated:

R. Kumar (✉)

Department of Computer Science, Meerut Institute of Engineering
and Technology, Meerut 250005, Uttar Pradesh, India
e-mail: mr.ravin_kumar@hotmail.com

© Springer Nature Singapore Pte Ltd. 2019

P. K. Kapur et al. (eds.), *System Performance and Management Analytics*,
Asset Analytics, https://doi.org/10.1007/978-981-10-7323-6_21

251

- 1.1 **Bubble sort:** The algorithm is slow compared to other algorithms. It has a complexity of $O(n^2)$ [4].
- 1.2 **Insertion sort:** Insertion sort is a sorting algorithm that works similar to sorting cards in hand. This algorithm also have the complexity of $O(n^2)$ [5].
- 1.3 **Merge sort:** This algorithm uses divide and conquer approach [6, 7]. It has a complexity of $O(n \log n)$.
- 1.4 **Quick sort:** Quick sort is more efficient than the preceding sorting algorithms. It has best-case complexity of $O(n \log n)$ and worst-case complexity of $O(n^2)$ [8].

Non-comparison-based sorting algorithms have less complexity because they do not require comparison operation among its elements to obtain the sorted sequence. Counting sort [9] is one classic algorithm used for non-comparison-based sorting. It has linear time complexity (i.e. $O(n)$), but is not preferred for sorting arrays having large numbers because it requires large amount of memory to obtain the sorted sequence. This paper presents a new approach of sorting which leads to a reduction of at least 50% of memory usage with a time complexity of $O(n)$.

2 Proposed Algorithm

Traditional counting sort has a time complexity of $O(n)$. The amount of memory required to perform sorting using counting sort depends primarily on the largest element present in the input array and less on the total number of elements in the sequence. In a sequence where ‘ T ’ is the largest number, It will require an additional array of size ‘ $T + 1$ ’ with index ranging from ‘0’ to ‘ T ’. For example, consider the case where we are sorting the values 1000 and 1 using traditional counting sort. Sorting these values require an additional array of size 1001 as the largest number in the input sequence is 1000.

Modified counting sort is designed to sort an array in the time complexity of $O(n)$, and also reducing the memory requirement of additional array by at least 50 percent. In the proposed algorithm, the details of parameters and variables are described in Table 1.

Table 1 Description of the parameters and variables used in modified counting sort

Parameter	Description
RA	Input array that is to be sorted
min and max	Represents minimum and maximum value respectively
RB and RC	Represents two arrays
Ref1 and Ref2	Represents minimum value in RB and RC respectively
RS	Represents the sorted array

Modified_Counting_Sort(RA)**BEGIN**

```

min =  $\infty$ 
max =  $-\infty$ 
/* min and max are calculated from RA */
for i = 0 to Length[ RA ] - 1
do
    if RA [ i ] > max
    then
        max = RA [ i ]
    end if
    if RA [ i ] < min
    then
        min = RA [ i ]
    end if
end for
div = (min + max) / 2
x = 0
y = 0
Ref1 = min
Ref2 =  $\infty$ 
RBmax =  $-\infty$       // RA is divided in two arrays
for i = 0 to Length[ RA ] - 1
do
    if RA [ i ] < div
    then
        /
* Ref1 is subtracted from all the values in RB, and max value of RB is
calculated and stored in RBmax */
        if RA [ i ] > RBmax
        then
            RBmax = RA [ i ]
        end if
        RB [ x ] = RA [ i ] - Ref1
        x ++
    else
        if RA [ i ] < Ref2
        then
            Ref2 = RA [ i ]      /* Ref2 of RC is calculated. */
        end if
        RC [ y ] = RA [ i ]
        y ++
    end if
end if

```

```

end for
for i = 0 to Length[ RC ] -1
do
    RC [ i ] = RC [ i ] - Ref2
end for

```

/* Now applying the New_Counting_Sort on both RB and RC sub arrays. */

New_Counting_Sort(RB, RS, RBmax - Ref1, Ref1, 0)

New_Counting_Sort(RC, RS, max - Ref2, Ref2, Length[RB])

END

New_Counting_Sort(A, B, k, Ref, AddLen)

A is the input array, B is the sorted array and k is the maximum number, which is present in A array.

BEGIN

```

for i = 0 to k
do
    C[i] = 0
end for
for j = 0 to Length[A] -1
do
    C [ A [ j ] ] = C [ A [ j ] ] + 1
end for
for i = 1 to k
do
    C [ i ] = C [ i ] + C [ i -1 ]
end for
for j = Length[A] - 1 down to 0
do
    B [ C [ A [ j ] ] + AddLen - 1 ] = A [ j ] + Ref
    C [ A [ j ] ] = C [ A [ j ] ] -1
end for

```

END

The time complexity of the modified counting sort is still $O(n)$ and it reduces the amount of memory required to perform sorting operation.

3 Working Demonstration

Consider the following input data set on which we have to perform sorting operation using the proposed algorithm.

20	33	23	60	52	210	240	200	130	42
----	----	----	----	----	-----	-----	-----	-----	----

On applying the proposed algorithm over this data, we get min = 20, max = 240 and div = 130. Values less than div are as follows:

20	33	23	60	52	42
----	----	----	----	----	----

Similarly, values greater than or equal to div are as follows:

210	240	200	130
-----	-----	-----	-----

From the algorithm, we get Ref1 = 20 and Ref2 = 130.
Now after subtracting Ref1 from first array, store the obtained data in RB. Similarly, after subtracting Ref2 from second array, store the obtained data in RC.
After performing above operation, elements in RB become

0	13	3	40	32	22
---	----	---	----	----	----

Similarly, elements present in RC become

80	110	70	0
----	-----	----	---

Now, our modified counting sort uses New_Counting_Sort on RB, which sorts RB and then adds the Ref1 value to the result of sorted RB:

0 + 20	3 + 20	13 + 20	22 + 20	32 + 20	40 + 20
--------	--------	---------	---------	---------	---------

Obtained result in RB:

20	23	33	42	52	60
----	----	----	----	----	----

And the size of C[] for doing this operation was 41 (i.e. from 0 to 40). Now, our modified counting sort uses New_Counting_Sort on RC, which sorts RC and then adds the Ref1 value to the result of sorted RC:

0 + 130	70 + 130	80 + 130	110 + 130
---------	----------	----------	-----------

Obtained result in RC:

130	200	210	240
-----	-----	-----	-----

Now, final result of sorting is obtained in RS, and the sorted sequence becomes:

20	23	33	42	52	60	130	200	210	240
----	----	----	----	----	----	-----	-----	-----	-----

Total memory required by additional C[] is 111, (i.e. 0–110), while on applying traditional counting sort, it would require an additional array of size 241, indexed from 0 to 240 to sort this input sequence.

4 Result and Discussion

Modified counting sort is tested on various input sets, and a comparison is done with the traditional counting sort algorithm. Since both algorithms have time complexity of $O(n)$, comparison is done on the basis of amount of memory required (Table 2).

To understand the effectiveness of the proposed algorithm the comparison using the above data set can be displayed graphically as shown in Fig. 1.

From the above data it is clear that in worst case, the memory used by the modified counting is half the memory used by the traditional counting sort algorithm. The time complexity of both the algorithms remains $O(n)$. This can also be graphically represented as (Fig. 2).

Table 2 Sample data to demonstrate the comparison between traditional counting sort and modified counting sort

Input elements	Memory of C[] in Counting sort (i.e. length of C [])	Memory of C[] in Modified_Counting_Sort (i.e. length of C [])
1, 2, 3, 250, 251, 252	253	3
18, 21, 19, 23, 407, 401, 402, 400	408	8
1000, 1001, 1002, 1003, 1004, 1005	1006	3
900, 800, 2, 1, 700, 300, 4, 7, 9, 12, 27	901	300
9, 8, 11, 10, 1001, 1700, 1500, 2000	2001	1000
20, 100, 1, 0	101	21
1000, 999, 998, 997 ...,3, 2, 1	1001	500

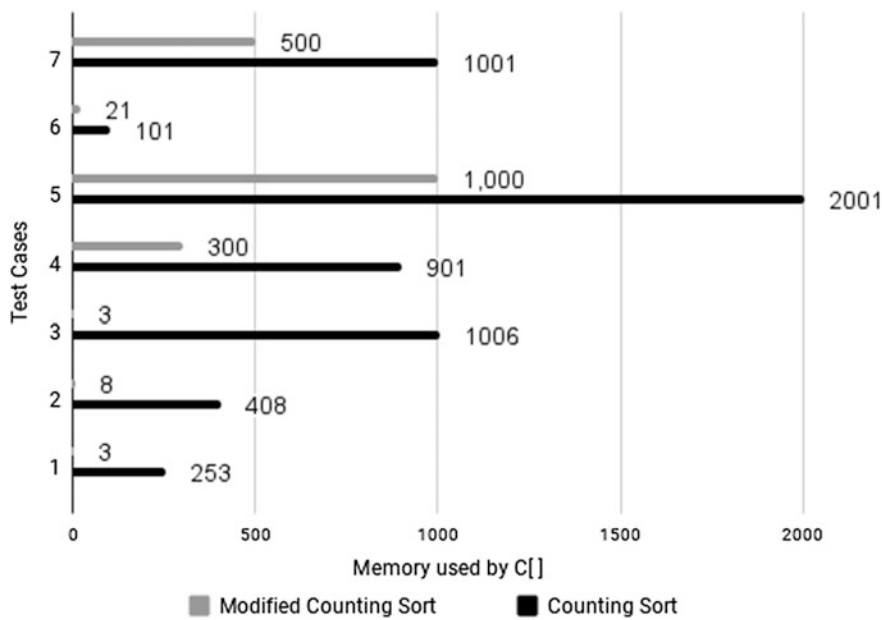
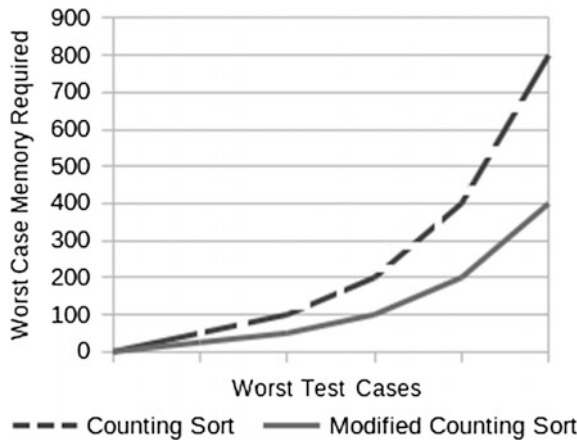


Fig. 1 Representation of memory used by each test data sequence in both traditional and modified counting sorts

Fig. 2 Representation of comparison of worst-case memory requirement by traditional and modified counting sorts



5 Conclusion

In this paper, an advanced method is introduced which reduces the memory requirements of the classic Counting sort algorithm by at least 50 percent. Modified Counting Sort can be used for improving performance of various existing sorting

algorithms like radix sort [10], which requires an additional stable sorting algorithm to perform sorting. Using modified counting sort, It is possible to improve the current sorting applications like searching a value in a large array, Sorting in large databases and TCP/IP packet sorting [11].

References

1. Flores, I. (1960). Analysis of internal computer sorting. *ACM*, 7(4), 389–409.
2. Franceschini, G., & Geffert, V. (2003). An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. In *Proceedings of 44th Annual IEEE Symposium on Foundations of Computer Science*, pp. 242–250.
3. Knuth, D. (1998). *The Art of Computer programming Sorting and Searching*, 2nd edn. Addison-Wesley.
4. Oyelami Olufemi Moses. (2009). Improving the performance of bubble sort using a modified diminishing increment sorting. *Scientific Research and Essay*, 4(8), 740–744.
5. Rupesh, S., Tarun, T., & Sweetes, S. (2009). Bidirectional expansion—insertion algorithm for sorting. In *Second International Conference on Emerging Trends in Engineering and Technology*, ICETET-09.
6. Radu, R., & Martin, R. *Automatic Parallelization of Divide and Conquer Algorithm” Laboratory of Computer Science*. Cambridge, MA, USA: Massachusetts Institute of Technology.
7. Dean, C. (2006). A simple expected running time analysis for randomized divide and conquer algorithms. *Computer Journal of Discrete Applied Mathematics*, 154(1), 15.
8. Friend, E. (1956). Sorting on electronic computer systems. *Computer Journal of ACM*, 3(3), 134168.
9. Rajasekhara Babu, M., Khalid, M., Sachin, S., Sunil, C., Babu, M. (2011). (IJCSIT) *International Journal of Computer Science and Information Technologies*, Vol. 2 (5) 2284–2287.
10. Andersson, A., & Nilsson, S. (1994). A new efficient radix sort. In *Proceedings of 35th Annual IEEE Symp. on Foundations of Computer Science*, pp. 714–721.
11. Meinel, C., & Sack, H. (2013). *Internetworking*. Berlin Heidelberg: X.media.publishing, Springer-Verlag. https://doi.org/10.1007/978-3-642-35392-5_2.